bash basics

What you should learn from this course part:

- use the terminal
- basic navigation
- simple loops
- basic Linux commands
- installations (if easy)

We type our instruction for the computer into the terminal, open one by typing (Ctrl + Alt + T) or by navigating to one using the GUI. In all languages used in the workshop, the # character indicates comments, and what follows will not be executed by the computer, but is there for the user.

```
#list the contents of the current directory
ls

#move to the home directory
cd
# list the contents
ls
# change to the directory with the data to be used
cd /home/schulung/Desktop/AB/db
# change to one directory up
cd ..
# make a directory and change into it
mkdir your_name_of_choice_here
cd your_name_of_choice_here
```

Alright, we can change between folders, look at their contents, etc... but we hardly needed to learn to use a command like to accomplish this. Let's very briefly look at somethings that are nice on the command line, like variables, loops, and wild cards.

```
## variables
# set one variable
species="Arabidopsis_thaliana_Col0"
# use (here simply display) the variable by adding a '$' before the name
echo $species
# that might already save some typing, or make it easy to perform
# a similar analysis on multiple different species. But variables are
# even more useful when it comes to loops
## loops
# this will loop through the numbers 1 - 20
for i in {01..20};
# the keyword 'do' designates the start of each iteration
do
# 'touch' will create an empty text file with the given name
touch my_sample_${i}.txt
# the keyword 'done' is used to designate the end of each iteration
done
# look at the result
ls
## wild cards
# we need one more file for the next example
touch keep_me_for_now.md
# note any differences in the results of the following commands
```

```
ls
ls *
ls *.md


ls my_sample_*
# now we can selectively clean up all those files we made
rm my_sample_*
ls # check result
# be careful with wild cards though, and when possible restrict
# their scope. Think about what would happen if you ran
# rm *
# in the wrong directory
## more on variables
# if you were wondering why we used ${i} and not $i in the loop
# the answer is it's best practice when the variable is used
# within words. Compare the results
echo $i
echo $isample.txt # '$isample' is not defined
echo ${i}sample.txt
# we'll also run into (and use) built-in variables
# for instance
echo $HOME
echo $USER
# we can return to our main directory this way
cd /home/schulung/Desktop/AB/
```

Now we will write our first shell script. We will try everything out before we actually make the script. There are five steps to making a script: (i) define what you want to happen, in our case, we want the computer to print the current time and date, (ii) test out the code by writing it out piece by piece, (iii) actually write the script using a text editor, (iv) make the script executable and (v) run the script.
First, test the code:

```
# make the computer repeat what you wrote
echo "My 1st script prints the date and time."
echo $(date +%F_%T)
```

And now we will produce the script (or program if you will). Open the text editor, gedit, either by double clicking a text file from the GUI file manager or via the applications menu under “Development”.
Then write the following in the text editor:

```
#!/bin/bash
echo "My 1st script prints the date and time."
echo $(date +%F_%T)
exit
```

Save the file as myfirstprogram.sh .
This program cannot yet be executed (ran) as the computer has not been told that it is an executable program. But we can set the permissions. We’ll start just by checking the current permissions.

```
# from the directory where you saved myfirstprogram.sh
ls
# note the colors
ls -l
# note the information you see on the screen
chmod u+x myfirstprogram.sh
# this command tells the computer to add (+) the
```

```
# execute permission (x) to the current user (u)
# if you omit the u, you will give permission to everyone
ls
ls -l
# compare what changed to output from above
```

As a side note, there's a lot more to permissions control, both in what can be done and also in how it can be done. It's more than we have time or need to get into here but there's a lot of nice resources out there to explain it if you're curious, e.g. https://www.pluralsight.com/blog/it-ops/linux-file-permissions.

You are ready to run the script.

```
# execute the script in the current directory (./)
./myfirstprogram.sh
# we can also store the output
./myfirstprogram.sh > text.txt
# look in the current directory in the GUI to find
# the file text.txt and open it by double clicking.
# further, view the output in the terminal
less text.txt
# enter 'q' to quit
# extra exercise:
# try appending output to the existing text.txt file
# by using '>>' instead of '>'
```

Writing shell scripts is easy, you just put in the program that you would write in the terminal and the computer will execute it line by line. The first line tells the computer what to use to read the program (in our case bash) and the last line tells it to exit from executing the program. Writing these small shell scripts becomes important when you want to run your read mapping overnight or over a weekend. We will revisit them when we do the read mapping.

You can also use commands to make analyses in a file. We will use a transcriptome (you will assemble your own later) which is in .fasta format

As necessary change into the directory /home/schulung/Desktop/AB/db

```
# look at the file using the command line
less Athaliana_primaryTranscripts.fa
# you can move through the document by pressing Enter
# you can leave the file by pressing q
# if you only want to look at the first few lines
head Athaliana_primaryTranscripts.fa
# if you want to look at the last few lines
tail Athaliana_primaryTranscripts.fa
# view the whole file
cat Athaliana_primaryTranscripts.fa
# now you know why the 'head' and 'tail' commands
# are so important. To interrupt this, you'll
# want the shortcut Ctrl + C (or Strg + C)
With these large files, we often get our first look and first stats in bash.
# count the number of sequences, more specifically:
# count the headers, which start with '>'
# the grep command, is a type of search, we'll look for '>'
# the '|' tells the computer to pass the output of one command
# to another, and wc is the command for "word count"
grep ">" Athaliana_primaryTranscripts.fa | wc --lines
# the number you see, is the number of transcripts
# you can also estimate* how many bases there are in total
# by counting all characters in the file with the
# exception of the fasta headers.
# you do that by inverting grep (-v), producing all but
# lines starting with ">", and then counting characters
grep ">" Athaliana_primaryTranscripts.fa -v | wc --chars
```

```
# *estimate, because the end-of-line character is counted.
```
Linux has many more such small useful commands. It is frequently helpful to google or to look at Linux tutorials and see what you can scavenge for your purpose.

Installations on a Linux system like Ubuntu can be very easy but also infinitely difficult depending on how well programs are programmed. In the best case, you can install by typing

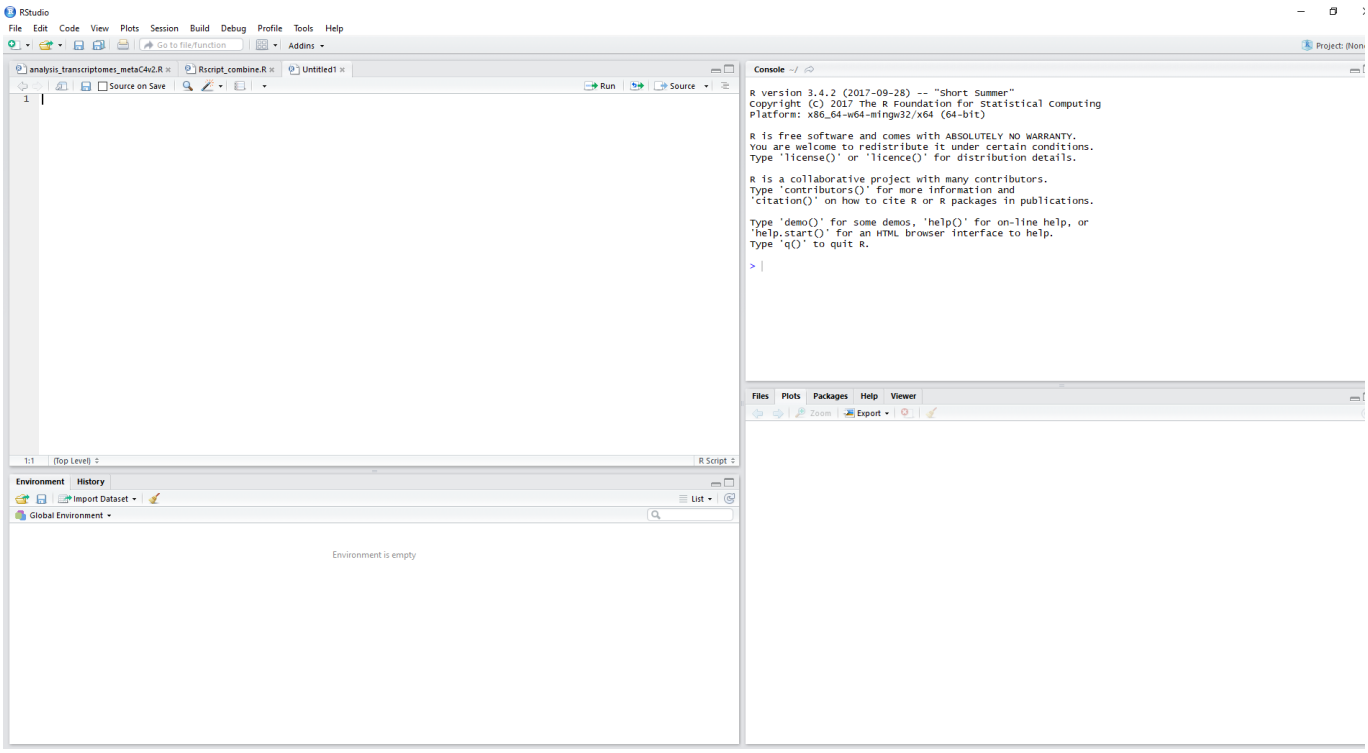
```
sudo apt-get install ncbi-blast+
```

which will install blast+ on your system. If your program of interest is not accessible for the package manager, the download and installation will be more complex. You will need to google it since we do not include it for time reasons.

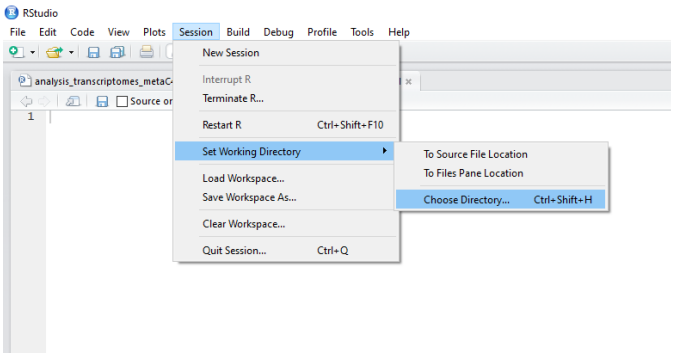
RStudio basics

RStudio is a developer environment for R (https://www.rstudio.com/). You can write a script (if you open a script) and execute the script (in the console) in the same desktop window. At the same time, the environment is displayed so you can see which variables are currently stored. A forth window conveniently displays the graphs and the help files if you call them up. Working in RStudio makes life much easier.

Open a new script by klicking on the green plus sign on the top left. Start writing your script line by line. Execute each line by (i) putting the cursor anywhere in that line, (ii) holding down CTRL and hitting enter. You will see that RStudio copies your code to the console and immediately executes.



You have to tell the program where your data is at and where you want to write, i.e. store the files that you generate during this R session. So choose the directory with the data.



What you should learn from this course part:

- get data into R
- manipulate data.frames (also known as tables in the real world)
- learn about the apply function
- learn the basics about ggplot2
- run through a simple RNA-seq analysis

The data you will import is simply a file with read counts for a control sample and a sample treated with Pseudomonas syringae.

```
dfr <- read.delim("raw.txt")
head(dfr)
#calculate rpm

dfr$Col.mock1_rpm <- (dfr$Col.mock.1/sum(dfr$Col.mock.1))*1000000
#calculate the rpms of the remaining samples

#LEARN HOW TO USE A FUNCTION AND HOW TO FORMAT DATA
#make principal component analysis
#it is critical to know what the functions you will use do
#it is critical to know how the data needs to be formatted

?prcomp
#google "pca prcomp r" to lear about how to use the function

#we need a data.frame (or matrix) which only contains the data to be used
colnames(dfr)
#you can choose your columns by their index number; the leading comma means column,
#a trailing comma would mean rows
df <- dfr[,c(8:13)]
#or choose by using the fact that they all have "rpm" in their column header
df <- dfr[, grep("rpm", colnames(dfr))]
head(df)
#why do I transform the dataframe?
tdf <- t(df)

#apply the function, make sure you know why I set scale=T
pca <- prcomp(tdf, scale=T)

#this will give you an error, try to read the error and understand what went wrong
#here I check if my idea of the problem is correct
table(apply(tdf,2,sd)==0)
#now I solve the problem and remove all columns which do not show variation
#you need to understand the following line to understand how to manipulate data in R!
#what does the comma mean?
#google and check apply functions
#what kind of vector is produced in the apply function == 0?
#what does the ! mean?
tdf <- tdf[, !(apply(tdf,2,sd)==0)]

pca <- prcomp(tdf, scale=T)
summary(pca)

#get the components out, make sure the output is a data.frame
scores <- data.frame(pca$x)
dim(scores)
head(scores)
```

```
#the next challenge is to plot data using R, here we want to plot the points with
ggplot

library(ggplot2)

#now let's work through plotting data

#tell ggplot where it can find the data and what to plot

ggplot(data=scores, aes(x=PC1, y=PC2))
#with this you get an empty coordinate system because ggplot did not know
#what you want on it
#add the points
ggplot(data=scores, aes(x=PC1, y=PC2))+
  geom_point()
#hm, what is what in this plot?
#let's change the data.frame with the information to include a label
scores$samplename <- row.names(scores)
ggplot(data=scores, aes(x=PC1, y=PC2))+
  geom_point()+
  geom_text(aes(label=samplename))
#you could now modify the sample name because we do not need the Col. or the _rpm
scores$samplename2 <- c("mock1", "mock2", "mock3", "Psm1", "Psm2", "Psm3")
ggplot(data=scores, aes(x=PC1, y=PC2))+
  geom_point()+
  geom_text(aes(label=samplename2))
#or you could choose to colorcode the labels instead of writing the label out
scores$treatment <- c(rep("mock", 3), rep("Psm", 3))
head(scores)
ggplot(data=scores, aes(x=PC1, y=PC2,color=treatment))+
  geom_point()

#now you can look at the cheatsheet for ggplot and figure out how to color the dots
#or how to change the background
ggplot(data=scores, aes(x=PC1, y=PC2,color=treatment))+
  geom_point()+
  scale_color_manual(values=c("black", "firebrick"))+
  theme_bw()
rm(df,tdf, scores, pca)

#ggplot acts by layering graphics on top of each other
#two key things are
#(i) have your data in order, i.e. more often than not, you will need a long format
#(ii) have your data in order!

#now, let's think more about data manipulation
#this is a simple RNA-seq experiment, so we want
#differential expression information
#fold-change information

#let`s start with the easy one: calculate the log2 fold-change
#first, get means
#look at the data again
head(dfr)
colnames(dfr)
#you can access the columns by column index
dfr$mock_mean <- apply(dfr[, c(8:10)],1,mean)
#of course, you can also access them using their column headers
grep("mock.*_rpm", colnames(dfr))
dfr$mock_mean <- apply(dfr[, grep("mock.*_rpm", colnames(dfr))],1,mean)

#do the same for the Psm treatment




#in both cases, we use an apply function to to the same thing to all entries in a
data.frame
```

```
#to do the log2 fold change, we define our own function
log2FC <- function(a,b) log2((b+1)/(a+1))
#and then we apply it to our data
colnames(dfr)
dfr$log2fold <- apply(dfr[, c(14,15)],1, function(x) log2FC(x[1],x[2]))
head(dfr)

#figuring out which are significant is more effort because
#we have to use a different package
#we install it first
if (!requireNamespace("BiocManager", quietly = TRUE))
  install.packages("BiocManager")
BiocManager::install("edgeR", version = "3.8")

#if the R version on the computers is older
#we might have to use
source("https://bioconductor.org/biocLite.R")
BiocInstaller::biocLite(c("edgeR"))

#in both cases we need the library
library(edgeR)
#first we choose the data to compare, raw data in our case
colnames(dfr)
df <- dfr[, c(2:7)]
head(df)
row.names(df) <- dfr$target_id

#now we have define where the replicates are
group <- factor(c(1,1,1,2,2,2))
group
#these are functions from the edgeR package executed as
#described in their documentation
y <- DGEList(counts=df,group=group)
y <- calcNormFactors(y)
y <- estimateCommonDisp(y, verbose=TRUE)
y <- estimateTagwiseDisp(y)
et <- exactTest(y)
#the next line extracts all the data
res <- topTags(et, n=Inf)

res <- as.data.frame(res)
table(res$FDR<0.01)
#finally we put the data together
dfr <- merge(dfr, res, by.x="target_id", by.y="row.names", all.x=T)
dfr$logFC <- NULL
dfr$logCPM <- NULL
dfr$PValue <- NULL
names(dfr)[names(dfr) =="FDR"] <- "FDR_Psmvsmock"
head(dfr)

rm(res,df, group, et,y)

#finally, we would like to export the table so we can
#look at it in other programs

write.table(dfr, file="mothertableV1.txt", sep="\t", row.names=F, quote=F)

#congratulations, you have finished your first (?) RNA-seq analysis based on raw data

head(dfr)
min(dfr$FDR_Psmvsmock)
#let's extract the top ten genes
#as always, doing easy stuff is difficult with programming languages
#while doing hard stuff is easy
library(tidyverse)

dfr <-dfr %>% arrange(FDR_Psmvsmock)
head(dfr)
topten <- dfr[c(1:10) ,]
topten
```

```
#if we want to plot those alone, we first need to pick our data and reformat from
wide to long
colnames(topten)
plot_tt <- topten[,c("target_id","mock_mean","Psm_mean")]
plot_tt <- gather(plot_tt,key="key", value="value", -target_id)
head(plot_tt)

#the data is now ready for a line plot


ggplot(data=plot_tt, aes(x=key, y=value, group=target_id))+
  geom_line()
#hm, I could now normalize the data, or just ignore one gene
ggplot(data=plot_tt, aes(x=key, y=value, group=target_id, color=target_id))+
  geom_line()+
  scale_y_continuous(limits=c(0,150))
```

Look back!

Do you remember how to read in tables and get tables back out of R? □

Have you observed how the data is constantly checked? head, dim, colnames? □

Did you understand the apply function? □

Did you understand the difference between long and wide format? □

Do you remember what the key parts of ggplot1 are? data, aes, geom? □

Now we will look at a second example with more data!

```
#clear your space
rm(list=ls())
gc()

#import another textfile, this one contains more information!
#this is data from Klepikova et al.; an expression atlas
#read the table "mothertableV1.txt" into a variable called dfr

#load data into R using a data.frame
dfr <- read.delim("mothertableV1.txt",stringsAsFactors = F)
#inspect the data.frame
head(dfr)
dim(dfr)

#How many rows does the data.frame have? Which biological unit corresponds to a row?
#How many columns does the data.frame have? Which information is stored in columns?

#calculate the means for all samples – before you start, consider how much work this
would be in #Excel

#we will learn how to loop over data in R


#first you need the names of all columns and you need to make them unique
colnames(dfr)
names <- colnames(dfr)
names <- gsub("_1", "",names)
names
length(names)
names <- unique(names)
names <- names[-1]
length(names)

#in the next step, we will program our first for loop, think through it, do not just
type it!
#to construct a loop we first work on one iteration and get our code done

i <- 1

#now we will test the content of what will become the loop later
#calculate the mean
means <- apply(dfr[grep(names[i],colnames(dfr))],1,mean)
means
dfr <- cbind(dfr,means)
colnames(dfr)[colnames(dfr) == "means"] <- paste0("mean_",names[i])

#okay, it worked
#now we remove column that we just generated and run our code as a loop
names[1]
dfr$mean_Cold.1.hour <- NULL

for (i in 1:length(names)){
  means <- apply(dfr[grep(names[i],colnames(dfr))],1,mean)
  #means
  dfr <- cbind(dfr,means)
  colnames(dfr)[colnames(dfr) == "means"] <- paste0("mean_",names[i])
}

#sometimes you want to merge in information from different sources
#for example, data from Mapman or from TAIR for easier understanding


mapman <- read.delim("MapManTAIR10.txt")
mapman <- mapman[,c(2,3)]
head(mapman)
#the gene ids are different! This will be a problem, so we need to alter the IDs.
dfr$target_id[1:10]
```

```
#let's test some code
str_sub(as.character(dfr$target_id[1:10]),1,9)

#and use the tested code for a new column
dfr$locus <- str_sub(as.character(dfr$target_id),1,9)
#now we merge in the information
dfr <- merge(dfr,mapman, by.x="locus", by.y="IDENTIFIER", all.x=T)
head(dfr)
dfr$NAME[1:10]
colnames(dfr)[colnames(dfr) == "NAME"] <- "mapman"

#sometimes you do not want the detailed mapman categories, just the top category
#as usual, we test code
temp <- sapply(str_split(dfr$mapman,"[.]"), `[`, 1)
temp
#now we use it to extend our table
dfr$mapmanCat1 <- sapply(str_split(dfr$mapman,"[.]"), `[`, 1)
#check out https://www.datacamp.com/community/tutorials/r-tutorial-apply-family to
#understand the family of apply functions in R
#this time we do not use apply, but a close relative sapply

#now try it for yourself: add the third, forth and fifth column
#from "TAIR10_functional_descriptions_20150630.txt" to the dataframe

#now we will learn to manipulate the table to pick particular groups of genes
#you can use the ones suggested in the code (photosynthesis) or use a group that
#interests you


dim(dfr)
#we exploit the tidyverse functions again:
ps <- dfr %>% filter(mapmanCat1 == "PS")
dim(ps)
#we can do the same by subsetting the dataframe rowwise in traditional language
ps2 <- dfr[grep("PS",dfr$mapmanCat1),]


colnames(dfr)

ggplot(data=forplot,                              aes(x=mean_Seedling.Cotyledons..S.C.,
y=mean_Leaf..mature..L.lg.))+
  geom_point(aes(color=PScolor,alpha=0.3))+
  scale_colour_manual(values = c("FALSE" = "grey80","TRUE" = "firebrick"))+
  geom_smooth(method="lm",color="black")+
  scale_x_log10()+
  scale_y_log10()+
  coord_cartesian(xlim=c(0.01,25000), ylim=c(0.01, 25000))+
  labs(title="leaf vs cotyledon", x="mean tpm cotyledon", y="mean tpm mature leaf")+
  theme_bw()


#Challenge:
#Try plotting flower against anthers, carpelle, sepal and petal
#and contrast them using a color code.

#plots for all samples together require reformatting the data!
#we will plot photosynthetic genes from multiple tissues
#choosing the tissues is just to make the plots easier to see and quicker to plot,
you can do all!

forplot2 <- dfr

#choose rows with PS genes
forplot2 <- forplot2[forplot2$mapmanCat1 == "PS" ,]
colnames(forplot2)
dim(forplot2)
forplot2 <- forplot2[, c(1,grep("mean", colnames(forplot2)))]
colnames(forplot2)
#maybe still too many for a plot side-by-side. choose a few you like!
```

```
forplot2 <- forplot2[, c(1, grep("Leaf", colnames(forplot2)),grep("Root", colnames(forplot2)) )]
colnames(forplot2)

#this is the key reformatting step!
forplot2g <- gather(forplot2, key="tissue",value="tpm", -locus)
head(forplot2g)


ggplot(data=forplot2g, aes(x=tissue, y=tpm))+
  geom_boxplot()+
  theme(axis.text.x = element_text(angle = 45, hjust = 1))

ggplot(data=forplot2g, aes(x=tissue, y=tpm))+
  geom_boxplot()+
  theme(axis.text.x = element_text(angle = 45, hjust = 1))+
  scale_y_log10()

#try some other geoms for plotting like violin plots or plotting jittered points
ggplot(data=forplot2g, aes(x=tissue, y=tpm))+
  geom_violin()+
  theme(axis.text.x = element_text(angle = 45, hjust = 1))+
  scale_y_log10()

ggplot(data=forplot2g, aes(x=tissue, y=tpm))+
  geom_jitter(alpha=0.3)+
  theme(axis.text.x = element_text(angle = 45, hjust = 1))+
  scale_y_log10()

#Challenge:
#Identify the the genes in which genes related to protein expression are very high (or low)
#and plot them as violin plot and box plot for a choice of tissues.


#now let's try to plot all the genes as line plots of expression
#see the group in the aesthetics!

ggplot(data=forplot2g, aes(x=tissue, y=tpm, group=locus))+
geom_line()+
theme(axis.text.x = element_text(angle = 45, hjust = 1))+
scale_y_log10()

#This way, you plot absolute data. Perhaps you want to plot scaled (i.e. z-scored) data. In this case, we need to go back to a dataformat in which the data is wide instead of long.
colnames(forplot2)
#when we reformat, we need to work with a matrix (all data types the same),
#not a data.frame

temp <- forplot2[,c(2:17)]
temp <- t(temp)
temp <- scale(temp)
temp <- t(temp)
forplot3 <- data.frame(cbind(forplot2$locus, temp))
head(forplot3)
colnames(forplot3)[1] <- "locus"
forplot3g <- gather(forplot3, key="tissue",value="tpm", -locus)
head(forplot2g)
ggplot(data=forplot3g, aes(x=tissue, y=tpm, group=locus))+
  geom_line()+
  theme(axis.text.x = element_text(angle = 45, hjust = 1))


#Challenge
#The file contains all genes, many tissues and three stresses
#Think about five biological questions and test them by plotting.
```